Tour Manager Protokoll

Übersicht

Die App ist unterteil in 5 Unterprojekte:

SWE_TourManager

SWE_TourManager.Models

SWE_TourManager.BusinessLayer

SWE_TourManager.DataAcessLayer / (TourManager.PostgressqlDataAcessLayer)

SWE_TourManager.Tests

Diese dienen zur Aufgabenteilung der verschiedenen Tasks und implementieren auch das MVVM Pattern:

Der „SWE_TourManager“ enthält die ViewModels und Views

„SWE_TourManager.Models entält die Models, auf welche wie VMs zugreifen

„SWE_TourManager.BusinessLayer“ enthält weitere Logikklassen, welche von den VMs aufgerufen werden

„SWE_TourManager.DataAcessLayer“ enthält die Datenbankfuntkionen und kommuniziert mit dem BusinessLayer

„SWE_TourManager.Tests“ entjäkt die NUnit Tests.

Klassen / Funktionen / Unit Tests

SWE_TourManager & Models:

Das Grundprojekt enthält alle Views welche zur Interaktion benötigt werden. Die MainWindowView.xaml ist hierbei das Hauptfenster, welches dann alle anderen benötigten Fenster (wie z.B. CreateTour.xaml) über einen Buttoncommand und dem dazugehörigen MainWindowViewModel aufruft. Alle Views besitzen ein ViewModel, welches die Hintergrundlogik der Buttons und Textbox-Variablen wie z.B. TourNames verwaltet.

Bei allem weiteren (Create Tours, Create Logs usw…) rufen diese VMs dann den BusinessLayer auf.

Models gibt es 3: TourItems, LogItems und MapInfos

BusinessLayer:

Immer wenn der BusinessLayer aufgerufen wird, gibt er entweder die Daten an den DataAcessLAyer weiter oder überprüft auf Richtigkeit.

Die wichtigste Klasse ist die TourManagerFactoryImpl, welche sich von dem gleichnamigen Interface ableitet und nur über ieinen Singleton-Kunstruktor verfügt (sollte es keine Instanz dieser Klasse geben wird eine erschaffen, sonst bekommt man nur diese Instanz zurück).

Außerdem implementiert diese Klasse das Factory-Pattern: Es werden Daten eingegeben und diese erstellt dann Entweder Tours oder Logs in kombination mit dem DAL und speichert sie in IEnumerable<ItemName> collections, welche dann wiederum von den VM als ObservableCollections zugreifbar gemacht werden.

Alle abfragen z.B. SearchTours werden über diese IEnumerables direkt geregelt, und werden beim Erststart auch über eine Abfrage des DALs erzeugt.

Eine weitere Klasse des BusinessLayers ist der Validator, welcher vor der Abfrage über den DAL überprüft ob die eingegebenen Werte überhaupt erlaubt sind. Dies wird über eine RegEx abfrage geregelt.

Die nächste Klasse ist dann der HTTPResponseHandler, welcher einen string über den DAL erhält und diesen in eine Anfrage der Static Map -Api umwandelt, also eine MapInfo zurückgibt.

Die letzte Klasse ist für das optional Bonus Feature. Der RandomTourBuilder gibt den Input-Aanforderungs entsprechende Random generierte Namen, zahlen oder Lat und Long Werte von Orten in Österreich zurück.

DataAcessLayer:

Der DataAccessLayer hat ein weiteres Unterprojekt, nämlich den PostgressqlDataAccessLayer, welcher die anfragen and die Datenbank übernimmt.

Der DAL hat zunächst einen DAO-Ordner, in welchen Interfaces für die Models auf Hinsicht der Datenbank definiert werden.

Desweiteren gibt es eine DALFactory welche ähnlich wie die TourManagerFactory DAOs erzeugt und eine Instanz der Schnittstelle zur Datenbank zurückgeben kann.

Die HTTPRequest Klasse bekommt Start und Endpunktdaten vom BusinessLayer, baut dann mithilfe der konfig gespeicherten API urls eine URI und sendet diese als Request los. Der erhaltene string wird dann zurückgegeben.

Der TourFileHandler regelt alle Imports und Exports und printed TourDaten ebenfalls als .txt Format.

Der TourImgHandler bekommt eine vom BusinessLayer erstellte MapInfo und lädt über eine weitere WebRequest (diesmal über die Static Map API) das Bild in einen vordefinierten Folder herunter.

Im Unterprojekt finden sich dann die Implementations der DAOs, welche dann explizit die SQL Commands als Strings gespeichert haben und über die Datenbank kommunizieren.

Tests:

Die Tests wurden über ein NUnit Projekt gehandhabt. Hier werden meist Klassen des DataAccessLayers (DAOs, HTTPRequest, TourFileHandler) getested. Dabei werden z.B. Abfragen über eine Id gemacht, welche in der Datenbank existiert, oder nicht. Desweiteren gibt es Tests zum BusinessLayer(Validator, TourManagerFactory), der Models (TourItem, LogItem) und ein Test über eine ClearAll funktion welche direkt im VM erledigt wird.

Lessons Learned

Da dieses das erste Mal war, dass ich ein Projekt in viele Unterprojekte und Layers unterteilt und einzeln programmiert habe, war es sehr schwer die Grundstrucktur zu schaffen. Die Videos, welche die FH zu dem Thema auf Youtube gestellt haben (WPFIntro und DALIntro) haben hierbei immens geholfen und meinen Zeitaufwand um ein vieles gesenkt.

Sobald jedoch das Grundgerüst stand war es sehr belohnend mit dieser Struktur zu arbeiten. Man konnte leicht weitere Funktionen hinzufügen und musste nicht z.B. für jedes WPF-Window alles neu schreiben.

Ein weiteres learning war es über eine config zu arbeiten. Dies war relativ schnell herausgefunden und machte auch sehr viel Sinn, um das Projekt auch auf anderen Systemen schnell zum laufen zu bringen. Es gab zwar eine Schweirigkeit das Testprojekt damit aufzusetzen, doch über den Forumseintrag war auch dies schnell gelöst.

Alles in Allem habe ich knapp 27 Stunden an dem Projekt gearbeitet, jedoch bin ich mir sicher, dass ich ohne die beiden Youtube Videos, welche ich in 4 Stunden abgearbeitet habe, bestimmt weitere 4 Stunden benötigt hätte, ehe ich die Grundstruktur funktionsfähig gehabt hätte.

Von den 27 Stunden waren:

4 Stunden Setup, Grunstrucktur des Projekts

6 Stunden Views + Referenzen auf die Models

6 Stunden BusinessLayer

7 Stunden DAL + Tests dieses Layers

4 Stunden Bugfixes, Designs, Research, bonus Feature, Protokoll

Git – Link: https://github.com/AngeloThau/SWE2